Escola Tècnica Superior
d'Enginyers de Telecomunicació
de Barcelona

PROJECTE FINAL DE CARRERA

SCATGK: Design, development and testing of a ticketing infrastructure for a 10k+ attendees non-profit Live Music Festival

GRADE: Enginyeria de Telecomunicacions (pla 92)

AUTHOR: Alex Barceló Cuerda
DIRECTOR: Oscar Esparza Martin

Departament de Telemàtica

Barcelona, 2016

A Telecogresca
Permeteu-me donar-vos
les gràcies en conjunt,
i tant sols afegir:
esca ESCA ESCA!

$+\alpha\rho$

Resum del Projecte

Aquest document presenta el disseny d'un sistema de *ticketing*. El disseny presentat no és exclusivament teòric: s'han desenvolupat aplicacions *software*, s'han construït prototips *hardware* i aquesta mateixa infraestructura s'ha utilitzat en events reals.

El projecte va néixer per a un propòsit petit i modest, i durant diversos anys s'ha anat millorant. Ara presento els detalls del disseny complet, amb especial atenció en el disseny i desenvolupament de l'*Access System* (Sistema d'Accés). No obstant, totes les parts de la infraestructura són analitzades: des del sistema de pagament *online* fins a la identificació de *tickets* dels usuaris.

En l'actualitat, la Infraestructura de *Ticketing* presentada s'ha utilitzat amb èxit per a un event de grans dimensions, la Telecogresca, a més a més d'haver estat provat i millorat gràcies a altres events de menors dimensions. SCATGK (el nom del Sistema d'Accés) ha demostrat ser robust i útil, a més d'oferir una sèrie d'avantatges respecte altres alternatives habituals.

Resumen del Proyecto

Este documento presenta el diseño de un sistema *ticketing*. El diseño presentado no es solamente teórico: se han desarrollado aplicaciones *software*, se han construido prototipos *hardware* y esta misma infraestructura se ha utilizado en eventos reales.

El proyecto nació para un pequeño y modesto propósito, y durante varios años ha ido mejorando. Ahora presento los detalles de diseño completo, con especial hincapié en el diseño y desarrollo del *Access System* (Sistema de Acceso). No obstante, todas las parted de la infraestructura son analizadas: desde el sistema de pago *online* hasta la identificación específica de *tickets* de usuarios.

En este momento, la Infraestructura de *Ticketing* presentada aquí se ha usado con éxito para un evento de grandes dimensiones, la Telecogresca, además de ser probado y mejorado a lo largo de otros eventos de menor envergadura. SCATGK (el nombre del Sistema de Acceso) ha desmotrado ser robusto y útil, en paralelo a ofrecer una serie de ventajas por delante de alternativas usuales.

Abstract

This document presents the design of a ticketing system. The design is not only theoretical: software application have been developed, hardware prototypes have been assembled and real events have used the proposed infrastructure.

The project was born for a small and modest purpose, and for several years it has been improved. Now I present here the details for the full design, with special highlights on the Access System design and development. Nevertheless, all the parts of the ticketing infrastructure are analysed: from the online payment system to the actual user ticket identification.

At the present date, the Ticketing Infrastructure presented here has successfully been used for a large Live Music Festival (Telecogresca), and has been tested and improved through several minor events. SCATGK (the name for the Access System) has proved itself robust and useful, while offering a handful of advantages over common alternatives.

CONTENTS

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Internet sales have become an increasing trend in a lot of areas. This phenomenon has changed the way companies sell things, and also the channels by which people buy goods and services.

I will focus this document on the Telecogresca Live Music Festival. This is a non-profit event organized by students, mainly from the Telecommunications Engineering school. It is an event with more than 35 years of history, and has evolved from a modest music concert near the university to a full-fledged Live Music Festival including international artists.

During the last years, the sales trend has shifted from the traditional "physical" channels towards internet sales. At first, existing commercial solutions were used. There were several companies providing a basic ticketing sales service in exchange of some fee to either the attendee or the organizer. More recently I have been involved in the foundation of our own Ticketing Infrastructure: SCATGK. SCATGK (*Sistema de Control d'Accessos Telecogresca*) is an in-house infrastructure which I will explain during this document.

Motivation

There is a first simple motivation: the fee. The commercial solutions charge for their service through the fee, and it can be quite substantial.

However, this is not the main motivation. It is true that we have the know-how for developing a ticketing infrastructure, but doing this effort must lead on to a significantly better solution than the commercial options. The costs associated to the development and deployment have to be taken into account. We (as members of the Telecogresca non-profit organitzation) do not get paid, but our time is not worthless.

Given all that, it is important to have clear and strong motivations, before devoting time and money to this project. There are indeed reasons to develop our own ticketing infrastructure:

Cash management The ticket management of Telecogresca moves a moderate-to-large quantity of money in a small time lapse (according to our standards). Having our own infrastructure allows us to manage directly the money flow, which may become critical for the organization of the festival.

Full integration The same ticketing infrastructure can be used for general festival access –this includes invitations, press passes, and so on. Using a single infrastructure eases the deployment and simplifies the management of all the attendees. This is something impossible to integrate with a commercial ticketing infrastructure, because a full integration requires a tailored application.

The challenge There is an extra motivation behind a lot of things that we, young engineers, do: It is because we can, because we want to learn, because we want to demonstrate we are capable of doing it. The Ticketing System was a challenge for me, and I learnt a lot (and I am still learning) during the process. This is not a main reason for developing the project, but it is a little bit of extra kick into it.

Structure of this document

This document is organized as follows:

This first chapter has covered the introduction of concepts and motivations. The objectives of the project have already been presented.

The available commercial solutions are presented and discussed in the following chapter, Chapter 2. Here I will expose considerations about their technical aspects and also the deployment logistics of those existing solutions. In addition to presenting those existing alternatives, the sale history and its trends is reviewed for several Telecogresca events (since 2010 up to 2016, both included).

In Chapter 3 the Sales Web Application is presented. The analysis on the design constraints is detailed at the beginning of this chapter. Qualitative use cases are showed, derived from the requirements. Finally, the technical details related to the design, development and deployment of the server-side site are listed. The sales application is –as expected– a fundamental entity for the ticketing system.

The Access System itself is analysed during Chapter 4. This part of the project includes the most time-critical aspect: the validation of the tickets during the music festival. The chapter explains the data acquisition process for the validation, and presents the validation strategy used. Annex I contains the full internal documentation related to the main application –the one detailed in section 4.1.3.

Once those two basic entities have been discussed, the Ticketing System core is completed. The Chapter 5 presents the design of this integration between those two components: the *Selling Web Application* and the *Access System*. Additionally, the deployment process (including a prototype developed for the Access System) is discussed.

Chapter 6 contains the conclusions of this project. There is a long path, from the initial idea of an in-house development of a Ticketing System to the final project presented in this document. This long path is completed with an analysis on the achievements and the future work for this project. Some of the more realistic and feasible future work is already being worked on, and it is also discussed in this chapter.

In the first appendix on this document (Appendix A) I will enter into some further detail regarding the *desktop application*. This application, first presented in chapter 4, is complex, important and full of technicalities. Reaching a stable version has been possible thanks to multiple iterations, a lot of improvements and several trial-error rounds. During this appendix I will explain some features and decision designs. However, even while going into some more technical programming aspects, the part maintains a high-level standpoint descriptive tone (as opposed to Annex I, where the tone is highly technical as it is an internal documentation, for programmers to programmers).

The Ticketing System requires an wide variety of skills and tools. There are a couple of appendix which contain further information of pre-existing frameworks that have been fundamental for this project:

- Appendix B gives a general description of the Yii web framework. This PHP framework is used in the Selling Web Application.

- Appendix C contains information on the Qt framework. The software application used at the Ticketing System relies on this C++ framework.

CHAPTER

2

PREVIOUS EXPERIENCE AND ALTERNATIVES

During this chapter I will present to the reader a general overview to some of the existing commercial solutions used by the Telecogresca Music Festival. The reader must consider that the presentation done in this document will try to be general and also that is based on our personal experience with those companies. Even if I tried to present a comprehensive analysis on those products, the commercial products may update and change. The explanation will remain in a "general overview" standpoint.

The ecosystem has been changing and evolving during the last years, the companies have shifted interests and the technology has improved. However, the basic mechanism of buying goods (in this case, tickets for a Live Music Festival) remains the same:

- The ticketing companies provide a friendly channel through which the buyer can acquire the tickets
- The organizer can manage those tickets.

It is worth mentioning another group of companies who provide special discounts, or some sort of restricted offers –like *Atrapalo* and *TresC*. However, those companies

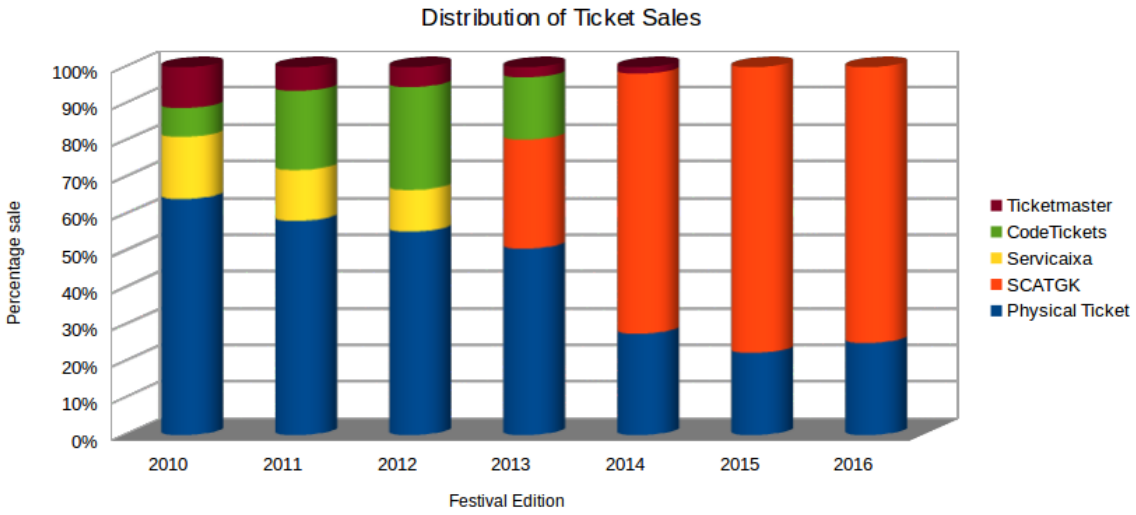


Figure 2.1: Historical per-vending platform ticket sales

tend to rely on pre-existing infrastructures and perform aggressive marketing strategies. This kind of service has little technical interest, so I will not present further details on them during this chapter.

The main platforms with which we have had some kind of business are:

- **TickTackTicket**
- **Ticketmaster**
- **Servicaixa**
- **Codetickets**

Those platforms may not be "separate" infrastructures, as there have been some rebranding amongst them and some partnerships, fusions...

If the reader is interested in the current state of those commercial solutions, the simplest and most accurate path would be to contact them and ask information as a potential client.

I include a brief percentage history on the ticket sales of Telecogresca, classified by vending platform, in figure 2.1. Later on, during this chapter (see 2.3), I will discuss in some more detail this figure and its meaning.

Ticket sale infrastructure

All platforms have some interface to introduce the ticket sale. The price should be included, of course, but they had some mechanisms to use extra features like quantity of available tickets, alerts, date spans, discount offers and other special scenarios.

The management of the ticket sale database depends from platform to platform. During one of our first implementations of our own infrastructure, Ticketmaster (the infrastructure we were working with during that year, back in 2014) provided us the list of valid tokens for all their sold tickets. This means that given a barcode we could easily –by performing a lookup into the list– decide if the ticket was valid. I used that to unify the validation of tickets by performing an adaptative read which recognized if a barcode was Ticketmaster or was ours, and performed the lookup to the corresponding database.

Even considering this successful use case of integration, their infrastructures were not really ready for that kind of deployment. I was able to adapt the application to their specific barcode configuration, which was not free of technical quirks. Generally, all the infrastructures expected us to use their own validation infrastructure.

Ticket validation infrastructure

The minimal entities required for a validation infrastructure would be:

Server Application which includes a database of tickets

Hardware ticket recognizer which has always been a barcode reader, as this is the main feature of the ticket that should be "recognized".

In all deployments which I have witnessed –from the previously mentioned ticketing companies– that is all the infrastructure provided by them. They provide a tight integration, because it is indeed their same service. The default validation infrastructure offered is very limited and do not allow easy integration nor customization.

The only exception I have witnessed to that "typical deployment" has been on the year that we deployed our own infrastructure, as explained in the previous section.

Very recently we had some contact with PlayPass, which I will describe as a validation infrastructure service provider. To name two services offered by that company:

- Automatic turnstile integration for unassisted festival entrance
- Cashless deployment for whole festival –including drinks and restoration

The features offered by PlayPass result in a much more appealing deployment, and is an infrastructure more close to what I will present during this document. However, while those services can be used by something like Primavera Sound, which is a 5-day, 100+ €-per-ticket festival, it is hard to argue for it as an "affordable proposal" for a single-day event like the Telecogresca Festival.

Our Experience and Recent History

In figure 2.1 one can see the brief distribution of sales since 2010.

First of all, the reader must note that it is a simplification of all our data, and it purposely does not take into account the total sales for each year. Instead of giving exact (and noisy) data, I will try to draw some conclusions upon the evolution of sales and the experience related to each of those years.

2010 is my first year in the Telecogresca non-profit organization. Strictly speaking, we sold tickets with TickTackTicket, not Ticketmaster, but this changed the year after and they currently are the same company. The main bulk of tickets were sold via physical channels. The "physical tickets" (the name I give to the tickets printed and sold by us) are more than 60%. On top of that, most sales of external infrastructures were indeed close to the "physical tickets": they were bought by physical people from physical people –e.g. FNAC is somewhat affiliated to Ticketmaster and most tickets sold that year included man-to-man interaction in the FNAC store. Servicaixa sales were commonly done through ATM machines instead of internet.

The main inertia from 2010 keep rolling for a couple of years. Ticketmaster and Servicaixa did not change a lot, but Codetickets became an attractive option for customers –less fees, more convenient checkout... The bars show the increase of sales through Codetickets and a decrease on physical tickets during the years 2011 and 2012.

Our own infrastructure officially launched on 2013. That year Telecogresca also kicked off two different kinds of tickets, with different prices. We discussed how to inaugurate our platform and the decision was to make an online limited edition of one of the ticket types, while the other type was sold by Ticketmaster and Codetickets.

The experience that year was a big success. The "limited edition" was loosen a little bit (not so "limited"), until I pressed to stop: I –and also some of my colleagues– wanted

a controlled and limited test scenario to check that everything was well-prepared. At the end, the infrastructure proved to be promising, and we learnt a lot of valuable experience. And we were ready to make it big.

2014 was the show off of our infrastructure. Physical tickets sales plumeted down in comparison to our shiny new online infrastructure. Ticketmaster was yet contracted, but that year validation infrastructure was merged with ours (as explained in previous section 2.2).

In one year (2013-2014), physical sales went from roughly 50% of total sales down to 25%. The SCATGK infrastructure, just a year after a limited test (less than 3 thousand tickets), absorbed more than 8 thousand tickets. It became the preferred vending channel for the Telecogresca Festival.

Although it may seem a sudden change, the tendency was already there. The online sales had been already growing for two years. A majority of the customers want to buy goods and services in their own pace. Once we offered our own online infrastructure with low fees and a friendly and easy checkout process, the customers moved onto it.

There is an interesting phenomena for 2016 (which cannot be seen in the graphic). We assumed that the fees played an important (almost critical) role for the adaptation of our platform, but in 2016 the fees were risen from 0.50 € to 1.40 €. We wanted to incentive the physical tickets while balancing all the derived costs of the infrastructure, previously covered by other budget items. The graphic shows no noticeable change of sales between this year and the previous one, proving us wrong on our first assumption. Albeit fees are important and can sometimes change the customer behaviour, its impact is not as strong as we initially supposed.

To conclude the analysis in this section I will say that the ease of use of the buying platform is *crucial* for success. Our own platform has allowed us to integrate tightly the sales with our web and the media campaigns. This, combined with nowadays customers trends, has proved the good adoption and suitability of the infrastructure presented in this document. But the reader must not forget that a Ticketing Infrastructure consists not only on sales, but also on validation, which I will explain in further detail during the following chapters.

CHAPTER

3

WEB APPLICATION

In this chapter I will discuss the characteristics of the web application. The web application is the "face" of the Ticketing System, as it is the first thing seen by the customer regarding the ticket.

First, I will present a comprehensive list of the requirements of the web application. After that, I will proceed to explain some technical aspects of the implementation. Finally, I present some qualitative considerations on the finished website.

Requirements

We want to achieve a highly customizable solution, so we manage ourselves the "shopping cart". Moreover, there is a low complexity in it, because there are few buying options.

However, we do not manage the actual payments. Managing credit card numbers is something outside the scope of this project (both in the technical aspect and in the legal aspect).

The main requirements of the web application will now be briefly presented:

Hosted and managed by us We, as a group of engineering students, are capable of maintaining the hosting and developing the web application. The customization capabilities of the final product should be backed by this.

Versatile There are various kinds of tickets, and the application can be used for other events. The price of the tickets may change (special promotions, discounts). The infrastructure should be able to cope with those changes, to minimize the adaptation costs between different events.

User friendly The web application is, indeed, a selling mechanism. All selling mechanism should be as simple and friendly as possible, to increase the chance of transaction.

Secure Both for the customer and for us. It is unacceptable that a user can steal a ticket from another customer. And we cannot afford to give "free tickets" due to a hole in the system.

Use Cases

Once we have established the basic requirements for the application, we can start developing the "use cases". First of all, I will review the basic usage from the point of view of the customer:

1. The (not yet) customer is looking the http://www.telecogresca.com/ website.
2. The customer decides to buy some tickets, so clicks on a link and goes into the Ticketing Web Application at https://entrades.telecogresca.com/.
3. He/She then chooses which tickets wants to buy.
4. Before paying, some personal information –basically the name for the tickets and the contact email– is gathered.
5. The system redirects to a standard, official and very secure payment gateway –outside of Telecogresca.
6. The customer inputs their credit card number, and proceeds to validate the payment.

7. The payment is done and the customer is redirected again to the ticketing web application.

8. The customer receives an e-mail saying that the payment has been validated. The customer can now download and print their tickets.

There are some details that are more easily illustrated with the point of view of the web application:

1. A visitor enters the realm and starts a buying process.

2. The visitor introduces the information. The payment process is stored and unique identifiers are generated.

3. Once the visitor is ready to pay, it is redirected to the external payment gateway, with the previous unique identifier. At this point, the system "loses" track of the customer. The payment is thus registered, but it is not considered paid.

4. The payment gateway sends (asynchronously, maybe delayed) a notification of valid payment for a certain payment identifier.

5. The database is updated, and an email is sent to the customer with a unique URL associated to the payment.

6. When the customer visits the URL, there are generated electronic tickets available. Each ticket has its own unique identifier (called *token*).

Framework

Developing a web application is a complex task, and there are lots of available tools which can aid in the process. Amongst all possible design decision, finally the framework chosen has been the *Yii framework* [1].

Just like most PHP frameworks openly available, *Yii* offers a mature and robust starting point for web applications. Simple abstractions to the database and the data models, plus a lot of extra libraries and tools, including some third-party support.

I present some details on this specific framework in appendix B. During the remaining of this section I will present a more high-level description of the design, almost framework-agnostic.

MVC design pattern

Nowadays, all web applications follow a design pattern called *Model-View-Controller* (**MVC**). This strategy is used because it divides the problem into several parts and decouples the complexity between them.

The concept of *Model-View-Controller* comes a long way, from the Smalltalk-80 [2]. However it has become ubiquitous in web applications [3]. In this section I will proceed to explain the key aspects of each of the *Model*, *View* and the *Controller* parts of our specific application.

Model

The *Model* is related to the actual data structures used internally. The web application is using a standard SQL relational database.

I will not cover the exact data fields used in the current software version, but here I will present the main features and information that can be found in the database.

The actual fields used in the application are too long and too especific to explain them there. Instead, I present a high-level explanation of the SQL tables used in the application.

TicketType For a certain festival Telecogresca tends to have several different types of ticket, with different prices and different constraints –e.g. a promotional discount only valid for the first 100 buyers, or only valid until a certain date. That information is stored in this table.

Tickets Every assistant to the Telecogresca has a valid entry in this table. Each ticket is related to a certain `TicketType`.

TicketsGroup Is the unit of sale. A certain `TicketsGroup` entry may contain more than one ticket, but only a payment must be tracked for each `TicketsGroup`.

Payment When a `TicketsGroup` is finished, an entry on this table is generated. The abstraction of final price is done at this model level.

Order This is very specific to RedSys, which is the payment gateway used internally. The basic idea is that a payment is attempted and control is driven onto the external payment gateway provider.

Response The external payment gateway provider sends a certain response through an asynchronous channel. That response is stored in this table –again, this is very specific to RedSys. A valid `Response` entry means that a certain `Order` has gone well which is associated to a `Payment` and thus to a `TicketsGroup`. The `TicketsGroup` being paid triggers the validation of all its related `Ticket` entries.

View

The *View* is the layout given to the final user (hopefully, the consumer). It is the visible face of the application and causes the first impression. The friendliness and usability of the application is directly related to a good design of the web page, which is what the *view* is for.

In figure 3.1 one can see one web page of the Telecogresca 2015 edition.

The web page showed is the first step on the buying process. The contents, the images, the layout, etc. is all part of the *view*.

From a technical point of view, we are relying heavily on CSS and PHP templates. Details on the CSS design is a topic related to web and graphical design. The PHP templates are heavily dependant on the controller and the framework used.

This part of the application has almost no technical content.

Controller

Model and *View* cover both the data structure and its display to the user. All the logic, session management, data processing and so on belongs to the *Controller*.

To oversimplify the explanation: the *Controller* is actually the "application" –with its source code and its execution. This is technically not strictly true, but it is a good simile for this scenario.

In this subsection, I will present the list and a brief description of their purpose. Let's present the public functions present on the `SiteController`, the main class.

```
1 class SiteController extends Controller {
2   public function actionIndex();
3   public function actionCompra();
4   public function actionVeure();
5   public function actionGetTpvForm();
6   public function actionError();
7   public function actionContacta();
```

Figure 3.1: First web page when buying tickets

```
8     public function actionCondicions();
9     public function actionPrivacitat();
10    public function actionLogin();
11    public function actionPdf();
12    public function actionPing();
13    public function actionLogout();
14    public function actionInvoice();
15 }
```

The exact syntax of those actions is Yii-dependent, but the main idea behind the actions is shared amongst all MVC frameworks. Each `action` is, more or less, mapped to a certain URL base. For example:
https://entrades.telecogresca.com/pdf?<token> will be attended by the action `actionPdf`.

`actionIndex` The index page, redirecting to `actionCompra`.

`actionCompra` Show the current tickets (do not show past promotions, do not show sold off tickets). Additionally, in case of a `POST` request, process the data sent by the potential buyer. item[`actionVeure`] Check a certain `token` (present in the URL) and show the tickets. This can be an ongoing invoice (just like a shopping-cart) or show already paid tickets, ready to print.

`actionGetTpvForm` This is a step between the web application and a secure external payment gateway. When the user wants to pay tickets (which have not yet been paid), they are redirected to external payment process.

`actionError` Management/display of errors.

`actionContacta` Static page for the contact information.

`actionPrivacitat` Static page for the privacy information.

`actionLogin` Admin login, for ticket management and statistics.

`actionPdf` Print the individual tickets of a valid payment. This `action` takes care of the PDF generation, which is done through a third-party library.

`actionPing` Process the asynchronous message from the payment gateway. This is not done by the user, but by the external credit card payment collector.

`actionLogout` Admin logout.

`actionInvoice` There are some cases where the user has already paid tickets but the tickets cannot yet be printed –like in special early promotions. The option to print the payment invoice is given to the user, and this action takes care of the generation of the invoice PDF –in a similar fashion to the `actionPdf` function.

CHAPTER

4

ACCESS SYSTEM

At this point, I have presented the web application. Typically, the buyer will only interact with the web application prior to the day of the festival –that is, assuming that everything goes smoothly. Once the attendee has purchased their tickets, they can print their tickets.

Moving on to the day of the festival, there is a need to validate the authenticity of those tickets. There are some common sense constraints:

- A ticket is valid only if it has been issued by the Ticketing Infrastructure.
- A ticket can only be used once.

In-field experience says that there will always be some kind of incidences. My personal experience during the deployment and testing of this project is that the end-user is the main source of problemes. Fortunately, the system has proven to be robust and the majority of attendees have successfully used the system and entered the festival. Later in section 5.2 I will elaborate on the incidences and their resolution.

There is an aspect that I cannot stress enough: the access system must allow a high steady incoming flow of attendees. This may not be a critic constraint in other Music Festivals or events, like the *Primavera Sound*. But it has proven to be a requirement

–and a pretty hard challenge– for our Telecogresca Music Festival. All the volunteers in Telecogresca have had a hard work of planning and improving the Access System infrastructure. I will not discuss all our previous experiences, but instead I present the current design of the Access System which has proven to be robust and yield a low incidence per attendee ratio.

I will divide the validation into two different categories: *frontend* and *backend*. The first one refers to the first and more physical step of the validation, and the *backend* validation refers to the logical and more technical process of authenticating the ticket. This division has allowed us to use different frontends, suiting them to our needs. In the following section, I will explain the characteristics of each of them.

Frontend validation

The *frontend* validation refers to the more physical interaction: a physical person handling a physical ticket to something or someone. After that, the system requires a *digitalization* process to translate the physical ticket into a certain digital value which will be handled by the *backend* validation. Note that the *digitalization* term used here has to be understood in its most broad sense.

Ticket (printed) information

Each attendee must have its own ticket, and the tickets contain the following fields:

Name of the attendee Mainly used to solve incidences, as it is not a reliable field.

Barcode This is the main information on the ticket. When the ticket is correctly printed, the barcode is digitalized into its value and the *backend* validation authenticates it. The standard used is the Code 128 [4], which is both versatile and broadly used.

Security code There are situations in which the previous barcode cannot be read –e.g., a bad print, a wet ticket. The security code must be digitalized by hand

The printable document has more text on it, like the ticket conditions and some extra legal information. However, all those informations are superfluous in terms of the validation and authentication of the ticket.

Figure 4.1 shows a sample ticket. On the left (figure 4.1a) the original ticket can be seen. The right image (4.1b) shows the typical artifacts and problems that damaged tickets show. To name a few of them:

(a) Original ticket

(b) Attendee actual ticket

Figure 4.1: Printed ticket sample

- No black ink or bad toner condition.
- Irrational folds, badly crossing over the barcodes.
- PDF artifacts rendering the barcodes useless.
- Water, alcohol, dirt or other stuff.
- Generally crumpled and damaged pages.
- Insufficient print margins.
- Ultra low quality.

Smartphone application

The first idea on the frontend validation was using smartphone devices. They are ubiquitous, are inherently wireless and they pack all the interface required:

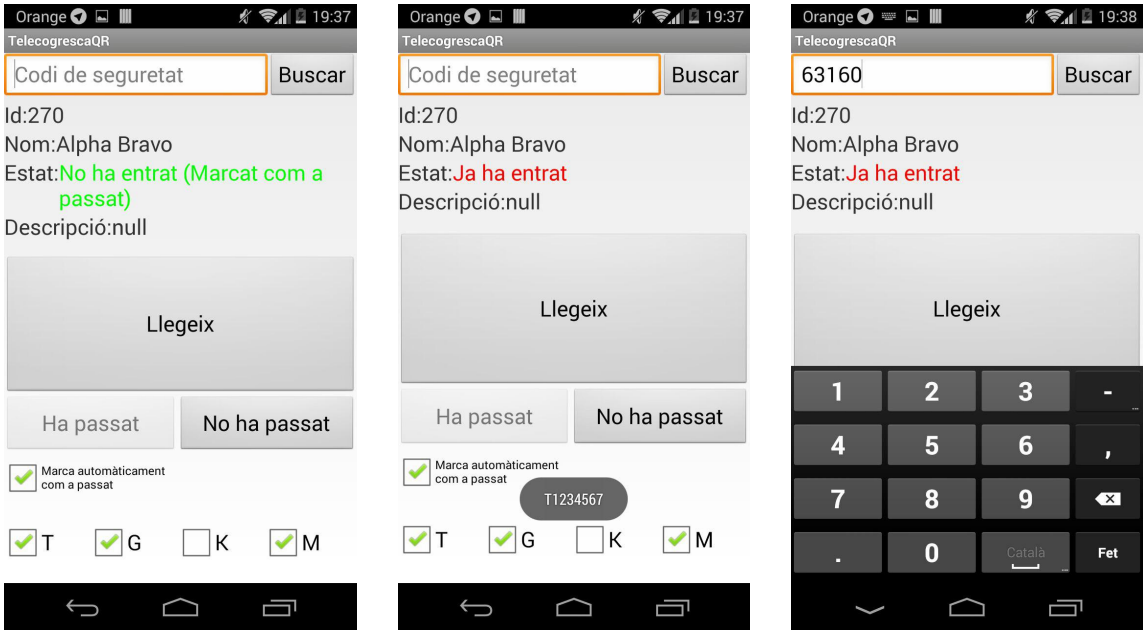
- Screen
- Camera
- Keyboard (touchpad)

We developed a Android application which was capable of be used in the Access System set up. Our experience has demonstrated that it was a valid application, although using smartphones was not an ideal decision. First I will show the basic characteristics and behaviour of the application, and then I will discuss some problems that arose from this configuration.

Some example screenshots of the user interface can be seen in the figure 4.2. The first one, figure 4.2a, shows the typical valid ticket. In this scenario the ticket is automatically validated and entry is granted to the ticket carrier.

The second screenshot, figure 4.2b, shows what happens when a user tries to use the same ticket twice. The Access System detects that the ticket has already been validated and warns the system operator about it. The operator should take appropriate measures.

Manual input of the Security Code input is shown in figure 4.2c. This can be used in a handful scenarios, but the most common one is badly printed tickets.



(a) Valid ticket (b) Invalid ticket (c) Manual entry

Figure 4.2: Examples of the user interface of the smartphone application

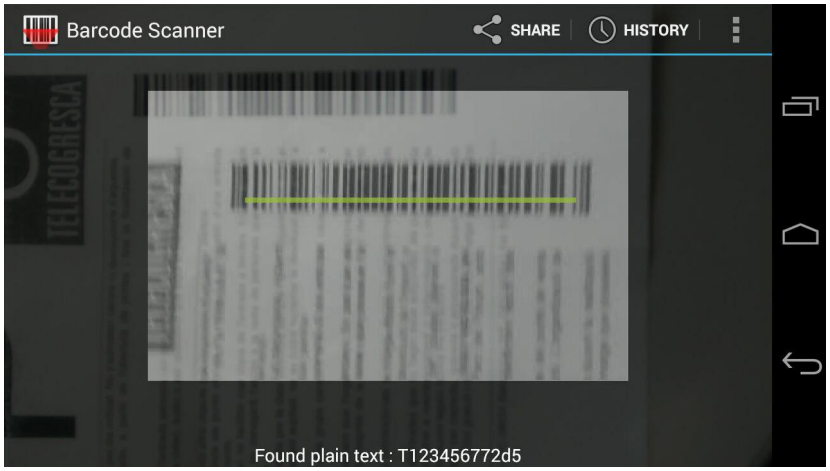


Figure 4.3: Barcode reading screen

The actual barcode reading is done through a third party application. Figure 4.3 shows this application called *Barcode Scanner* (available free-of-charge through the official Android Play Store).

This mobile application was useful to test the infrastructure and also to design and improve the backend. However, when tested "in the field" it proved to have some major drawbacks. First of all, the mobile camera has low sensitivity and depends a lot on external illumination, which is a problem for outside deployments. The battery life is always an issue, so using the flash light is an unacceptable solution. The speed and responsiveness of the mobile was enough in some devices, but was unacceptable for the low-end Android devices. Also, those devices –common ones– are fragile and cannot be easily anchored to a place. If the mobiles are left unattended, they tend to be either stolen or vandalised.

A new frontend application was required, and I decided to start a more complex *desktop application* which could be deployed in a more enduring fashion.

Desktop application

The *phone application* is a versatile solution which packs all the interface required by the system, but some of the flaws and drawbacks inherent to the phone devices makes this phone-based solution unfeasible for a music festival deployment.

Using a *desktop application* allows us to use commodity hardware and adapt it to our requirements. The PC needs a screen –or some visual output– to indicate the validation status of the ticket, and to receive the barcodes as inputs. A keyboard is a nice thing –we need them, under certain scenarios, to input the Security Code– but it is not required everywhere.

Figure 4.4 shows the organization of lanes and hardware –in this case, a laptop– when using desktop application for validation.

In this configuration, we can see at the figure that for each laptop there are two "validators". Each "validator" is actually a person that performs the ticket validation –typically by digitalization through a barcode scanner.

The attendees go through the queue, get their tickets validated, and enter the festival.

Each validator person is assigned a lane, and thus uses a certain half of the screen. This allows to make the most of the laptops –which have always been a scarce resource in our deployments– while being feasible to deploy and use.

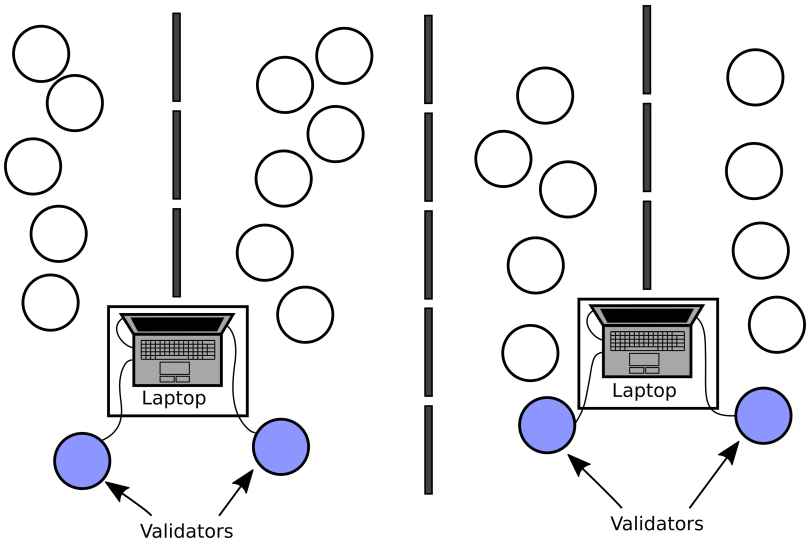


Figure 4.4: Sketch of the ticket validation lanes at the festival

Assigning more than two lanes per laptop has been considered and tested. The first deployment setup consisted in a hardware device (which will be reviewed in 5.3) which allowed two screens and up to 8 lanes. It was a clean, compact, easily deployed piece of hardware which we hoped it would allow a high throughput in terms of validation. However, it proved completely useless in that regard, as the visual reach of the validator tends to be one to two lanes. The mentioned deployment required a visual reach of 4 lanes, which was ridiculous. In addition to that, the deployment required a high ratio of wireless barcode readers, which are more expensive and less reliable than their wired counterparts.

Fortunately, we realized this problem and quickly simplified the software application to allow two lanes per device. This solution requires more hardware devices, but has proved to be robust and easy to use by the validators.

The basic visual aspect of the designed application can be seen in figure 4.5, just when it has been started. Figure 4.6 shows a certain sample state of the application, and the annotations of the meaning of each part can be seen in figure 4.7.

The shown application is programmed in C++ and uses Qt [5] as the graphical framework. I decided to use this combination of language and graphical framework because it is a reasonable trade-off between performance and features, and the application is easy to port between architectures.

For some basic information about the Qt graphical toolkit see Appendix C, which includes some basic characteristics and a comprehensive list of features.

Figure 4.5: Desktop application

Figure 4.6: Sample state of the desktop application

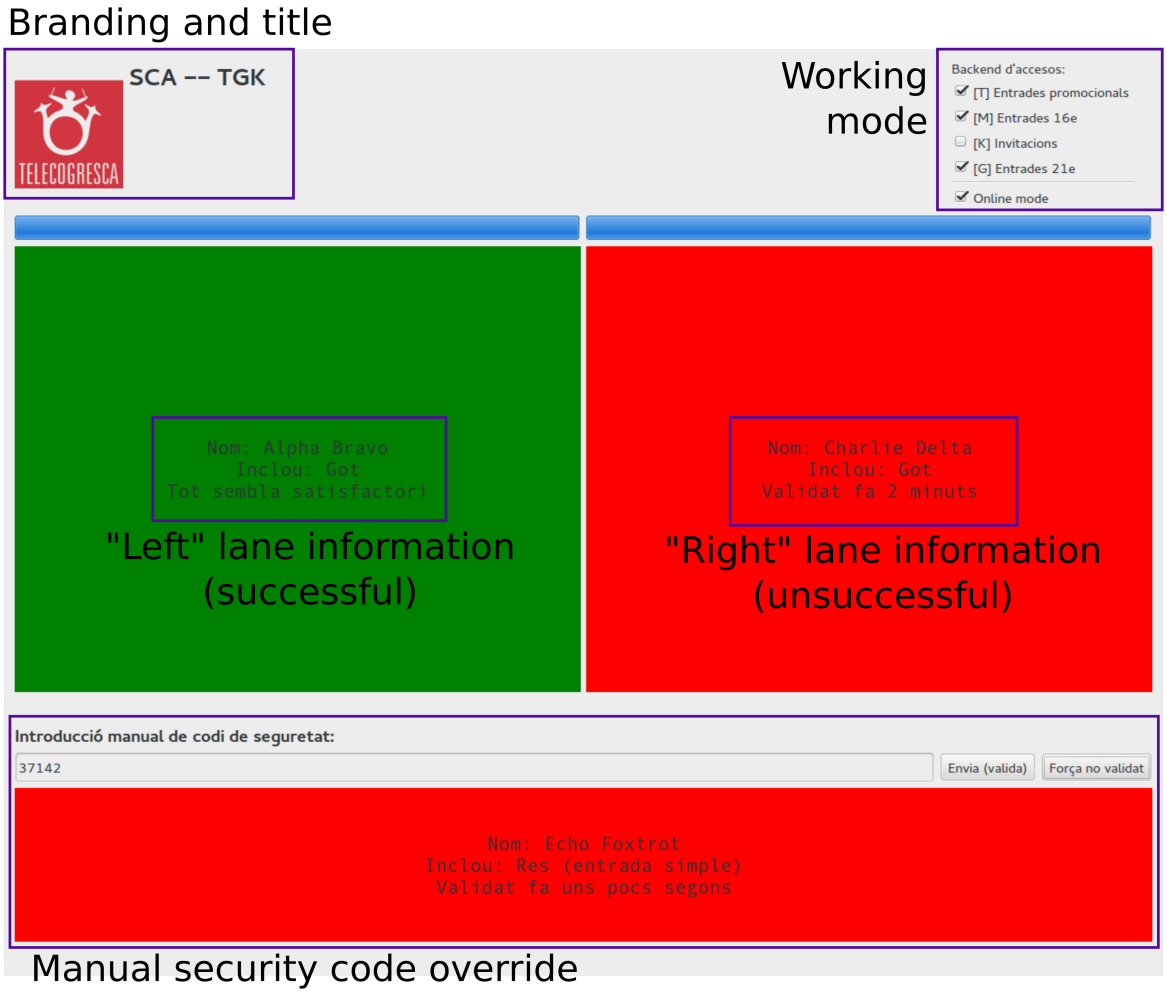


Figure 4.7: Description of interface main blocks

Looking at the annotated figure 4.7 we can see that roughly each half of the screen is dedicated to each lane. There is some vanity branding on top –although this application is not visible to the attendees, only to the validator volunteers.

There are two extra spaces in the application's main window that I have not discussed yet. First, the *Working mode*, allows to choose which tickets are allowed in a certain validation place. This proves to be very useful when having VIP lanes, or wanting to segregate different types of tickets. It also allows to reuse the application between completely different scenarios. One example of that is the usage of the same application in the entrance and in the "drink exchange", where people who bought special tickets could go to get their corresponding coupons.

Another feature of the application is the *Manual security code override*. There is only one place for it because it needs an actual keyboard, and I considered the scenario where there would be only one physical keyboard per device. This input can be used to enter the security code for tickets that cannot be barcode-read –this happens with badly printed or damaged tickets. The application allows to enter a security code without halting any lane. We (wrongly) assumed this to be unimportant, and the security code input was not available in the first version. We quickly realised the mistake, as the bottleneck of the whole system became those "bad tickets". This feature, added to a more intelligent design of the lanes and the attendees flow, gave us a significant boost in the validation throughput.

I think that further explanation about the desktop application is relevant, both due the amount of man-hours dedicated to it *and* the importance of it during the festival. However, explaining all those details here would break the big picture of this chapter. For those reasons, the reader is welcome to read the high level description in Appendix A and also the technical documentation available in the *Annex I – Access Control System (SCATGK)*.

Backend validation

The backend contains all the tokens of the tickets that have been sold. Moreover, some extra information –at least the one present on the printed ticket– should be available, in order to detect and manage forged tickets and any attempts to cheat the system.

The basic core of the backend is the database. The same database used in the Web Application (see previous chapter 3) is used inside the backend. This allows easy migration from the sales application onto the backend of the Access System. This

	Test	**Set**	**Test-and-Set**
Invalid token	Invalid Token [**e**]	Invalid Token [**e**]	Invalid Token [**e**]
Used ticket	Validated	Success	Already used [**e**]
Valid ticket	Valid	Success	Success

[**e**]: Error reply

Table 4.1: Backend responses

means that the backend deployment includes a SQL database with all the required information.

Using a database is a key point of the backend. However, a standalone database as the whole validation backend mechanism is not ideal. That would mean coupling a lot the frontend to the data model used, and repeating a lot of code between different frontends. The idea was to use some standard layer between the data model (the database) and the frontends (the application layer). The technological discussion about the integration of the backend validation will be addressed in chapter 5.

The backend must allow two basic mechanisms: queries (*test*) and modifications (*set*). In addition to that two elemental operations, a basic atomic *test-and-set* is implemented for the trivial use-case: a ticket-holder tries to enter the festival. Note that the set can generally be used for both validation of a ticket on entering and de-validation of it, although that de-validation process is only used in extraordinary situations. Table 4.1 shows the responses issued by the backend for all the combinations of ticket state and operation requested.

CHAPTER

5

SCATGK INTEGRATION AND DEPLOYMENT

During the two previous chapters I have already presented the main pieces of the project:

Web Application Which is used to buy tickets –previously to the event.

Access System Which is used to validate the tickets –during the event.

In this document, I have highlighted more the technical decisions and the design of the *Access System* because I was the person responsible for it, thus it is the area I am more proficient on.

There is an extra effort that I will explain and discuss during this chapter: integration & deployment. This project explains an infrastructure that has been field-tested and validated in real events, and until now I have focused on its two main building blocks. Now I will proceed to explain some ways used in order to combine those parts, both in the software level and in the hardware level.

Rigorously speaking there is no deep need of *integration* between the two parts that I have described previously in this document. They both can, more or less, work

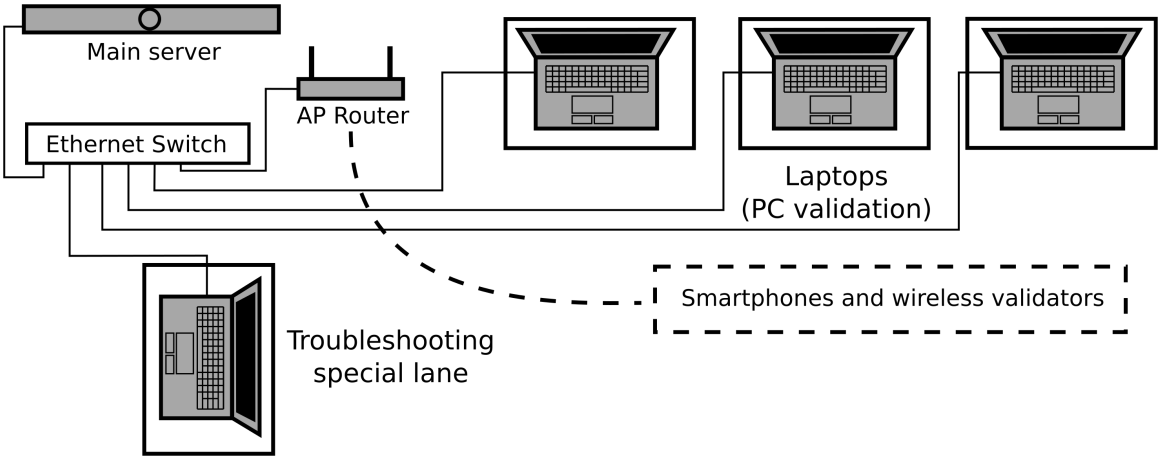


Figure 5.1: Network layout

autonomously –which is, in fact, an approach used by a lot of ticketing companies, as presented in 2.2.

Once said that, I have to remind that one of the goals was to achieve a *full integration*, and also that *integration* is not a trivial matter. In this section I will explain some mechanisms that we introduced in our design and in our software stack in order to achieve the desired coupling between the infrastructure blocks.

Communication

The first thing to discuss about the deployment is the communication between the different hardware devices. There are two main sides: the "data container" –fancy name for the machine that will serve the main database– and the validation endpoints.

Communication network layout

A rough draft of the typical network layout of the festival can be seen in figure 5.1.

The connections are pretty straightforward: there is a central main server, which contains the database and performs validation. A wired ethernet network is used in order to interconnect the different validation endpoints.

In the figure I differentiate the main *PC validation*, which uses the Desktop application presented in section 4.1.3, from the *Troubleshooting special lane*. A special

validation endpoint is considered in order to perform troubleshooting. This endpoint should be managed by a SCATGK-knowledgeable person, as it has direct access to the data models and can perform extensive internal validation.

I will discuss more about the *troubleshooting lane* later in this chapter in section 5.2

Communication protocol

Up to now I have discussed the characteristics of the *frontend validation* and introduced the *backend validation* (see sections 4.1 and 4.2). In this part of the document I will elaborate on the technological matters related to the backend and the communication protocol.

RESTful API

The proposed backend implements a **RESTful API** [6](*REpresentational State Transfer Application Programming Interface*) through the `http` protocol. Using the RESTful architecture when developing web applications is a common technique. It generally allows reuse of code between the Web Application and the API, which is generally desirable and we have taken advantage of.

I will present a brief explanation of the philosophy behind the REST pattern, and discuss why it matches the requirements of the communication required in the integration.

First of all, a RESTful API is typically implemented over an HTTP channel. This allows the presented integration to use the same web server, and –in our case– allows to easily provide both an admin interface via a web browser and a RESTful tailored API through the same web server. This is an advantage *in addition* to code reuse. So it is a good design decision, both from the development and the deployment point of view.

Going more into the formal **constraints** of the architecture, there are two that are specially relevant in our API use case:

Client-server Clients do not have any data storage and do not take part in the server work. We have a clear separation between the validators (which become *clients* in the presented REST architecture) and server. This separation is both trivially satisfied and sensible.

Stateless The most common case of communication is the atomic *test-and-set* (check section 4.2), where a *state* is not needed. Being stateless allows the RESTful API to be simple both in its formal definition and in its actual source code.

The troubleshooting lane

I could present multiple anecdotes, in first person view, about the *troubleshooting lane*. We can divide those anecdotes in three big categories: drunk ticketholders, cheating attempts and bad reading comprehension. However, the important and useful conclusion is: there will be issues, there will be cheaters and you have to be prepared to cope with spurious but long troubleshooting.

The empirical rough estimation is that near 1% of tickets sold[1] have some issues in the festival entrance. During the first year, that ratio was higher and the time consumed by those tickets was too long. The current infrastructure can cope with all those special cases and solve them in a fast fashion. This is due to a mix between better management interfaces at the troubleshooting lane and clearer action procedures for common problems.

Hardware prototype

The first prototype that I developed consisted in a high-density package capable of holding two independent instances of the application, with one screen each and 4 control lanes per screen. The "yield" of this prototype is, therefore, 8 lanes.

After some field-tests, the experience showed us that this was not a great idea. As already stated in 4.1.3, the visual reach of the validators (given that the "validators" are human beings) was not compatible with this deployment. Also, wireless barcode scanners proved to be somewhat inconvenient at times, so this design has some critical flaws.

Once said that, the insight that I gained from building this hardware prototype is something that I value a lot, and it helped me –and also some other members

[1] The reader must take into account that 1% of ten thousand tickets equals 100. That means that in a typical Telecogresca festival, the Telecogresca's troubleshooting volunteer must face 100 people that will be either trying to cheat, completely drunk, or unable to understand the basic ticket conditions. During the last two years, that volunteer has been me. I must say that 100 people is a lot of people, and being in this lane is somehow a fast way to lose a little bit of faith in humanity.

of Telecogresca– to understand the problems and attain better designs and cleverer solutions.

I will present here the rough steps followed in order to build this hardware prototype.

Bill of Materials

- 2x Raspberry Pi
- 2x Powered USB Hub (4 connections in one side, at least)
- 1x Small-sized ethernet switch (no need for gigabit, but should be powered with a 5V wall-adapter)
- 1x 5V power supply
- Aluminum case big enough for all the previous components

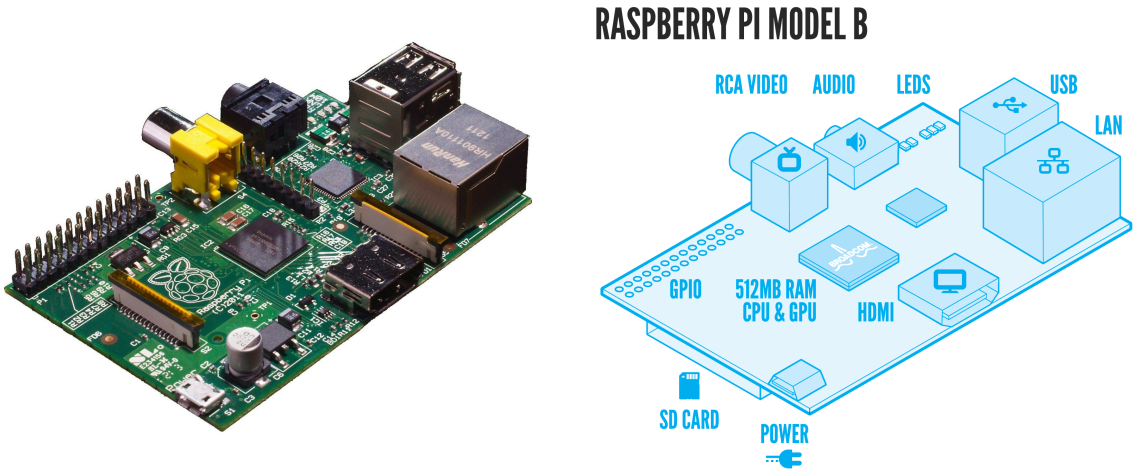
And, of course, time, patience, cables, screws & nuts. Also: connectors and basic soldering skills.

About the Raspberry and its peripherals

The Raspberry Pi is a "credit-card sized computer" [7]. It is a single-board computer based on a System-on-Chip processor. It is a simple, low power, low performance cheap computer. The flavour used in this project is the "Raspberry Pi B". The onboard ports that this project focus on are: two USB ports, an Ethernet NIC, and HDMI video output. See figure 5.2a for an image of the actual hardware, and figure 5.2b for a diagram of the placement of the onboard ports.

Using Raspberry Pi devices allows to pack not one but *two* "computers" in a small factor form. It has a nice price tag: ∼30€. It runs Linux (Raspbian, which is a Debian-based distribution) and a lot of packages can be easily installed without hassle.

The Raspberry has regular Ethernet network capabilities. Having two separate Ethernet cables seemed a bad decision –given that the deployment is already a delicate matter and adding cables is never good news. I looked into form factor of generic network switches and it seemed very easy to disassemble one and adapt to my needs, and it indeed was. Using a 5V Ethernet switch allowed us to use a single power supply for all the components.



(a) Device photograph

(b) Onboard ports schematic

Figure 5.2: Raspberry Pi figures

The Raspberry Pi did not have enough USB connections and they are oddly placed and they tend to not provide enough current for some peripherals. The solution is easy: use a powered USB hub. Given the lack of speed constraints for our peripherals –they are plain keyboards, at the eyes of the USB protocol– I was able to obtain cheap hubs and use them into the prototype. Again, using a 5V hub –which most of them are– allowed to share the same power supply.

By using those components the final result was a compact box which was capable of holding two computers, each of them with a screen –not included in the box, but through a HDMI connector of the Raspberry. The network entered through a single cable, and enough USB connectors were available without fear of over-draining current from the Raspberry.

Building the prototype

I will avoid specific details of the actual building, but here I will (briefly) describe the finished build. I believe that the final prototype –although not a successful and long-lasting device– draws some interest, but all the details, troubleshooting, problems and improvements do not provide any deep understanding to the matter.

First of all, I present a high-level schematic in figure 5.3 to outlay all the internal connections. This figure shows all the different wiring regarding the prototype:

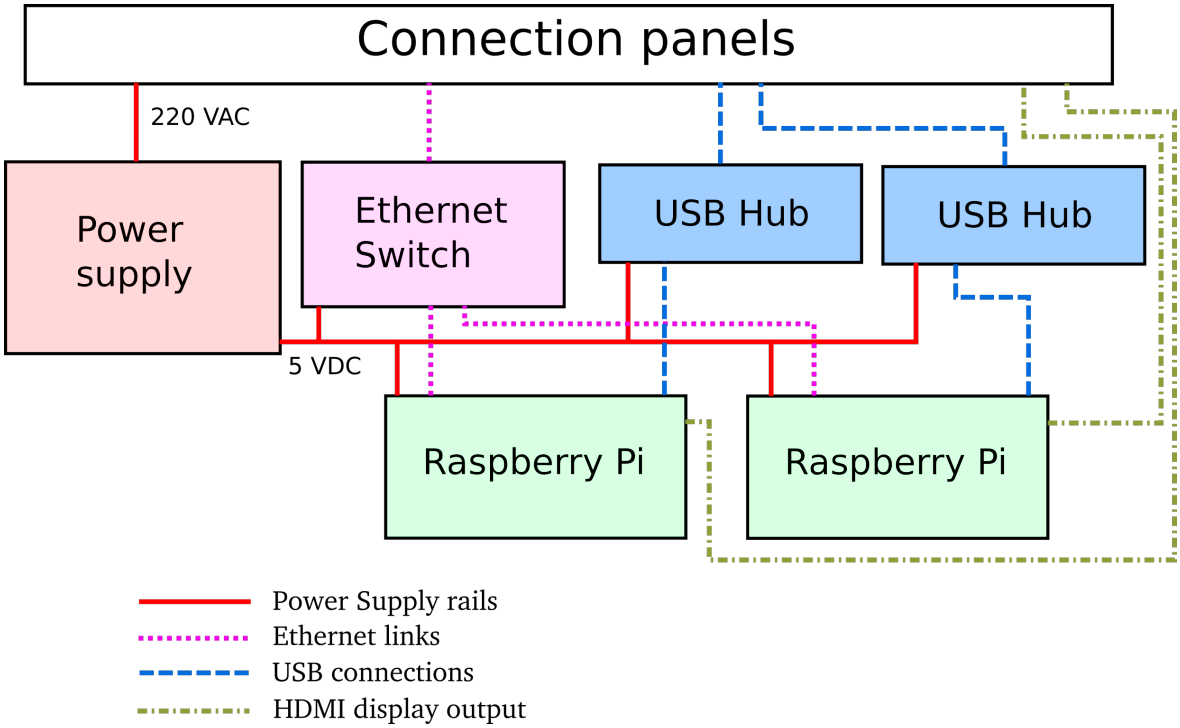
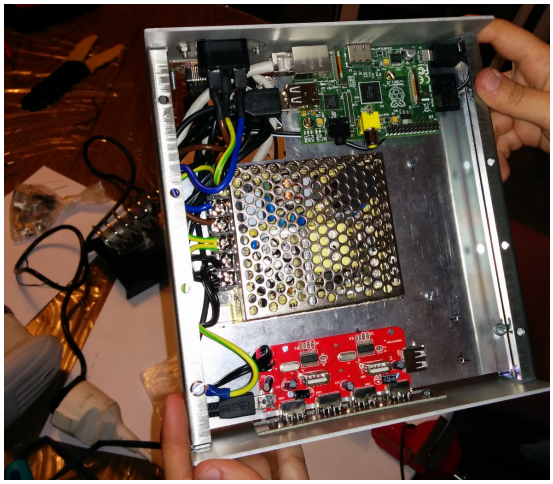


Figure 5.3: Global schematic of the internal connections

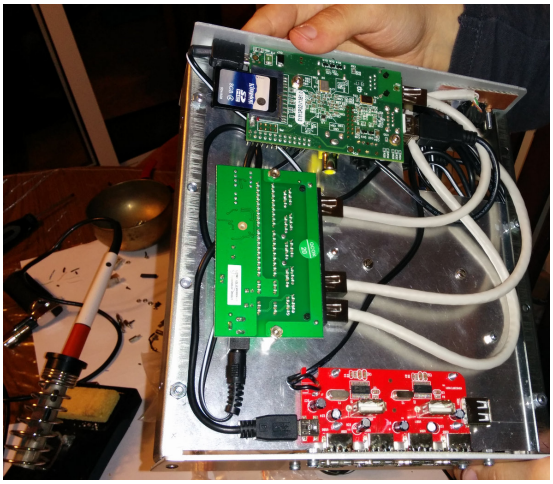
- Power supply
- Ethernet (network)
- USB
- HDMI (display)

The inner components can be seen in figures 5.4a and 5.4b. The components, previously listed in the "Bill of Materials" (5.3.1) can be identified with the figures 5.4c and 5.4d.

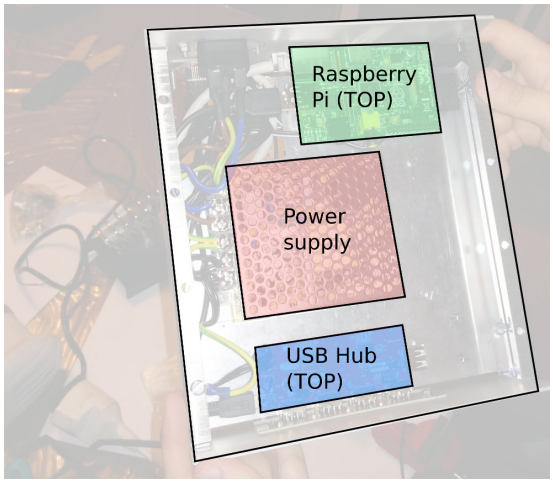
The aluminum box used is 200x220x80mm (WxLxH). The components are not cramped, but there is not a lot of extra room available. All the connections are available through the front and rear panels (figures 5.5a and 5.5b).



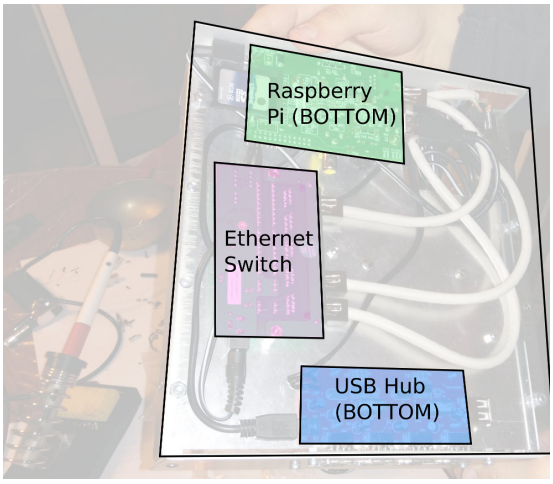
(a) Top layer components



(b) Bottom layer components

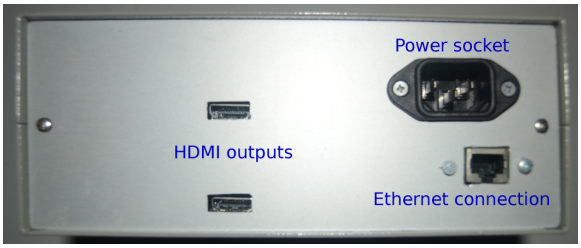


(c) Top layout

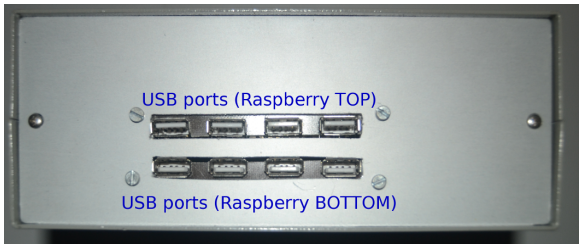


(d) Bottom layout

Figure 5.4: Pictures of the hardware prototype, and component layout



(a) Front panel



(b) Rear panel

Figure 5.5: Prototype connection panels

CHAPTER

6

CONCLUSIONS

This project has become a very important asset for the Telecogresca. I am very proud of all the work done on it, along all of my colleagues who have coded, worked, tested and improved it.

We can say that the application has improved a lot since its inception, and it currently works as expected and has nothing to envy to its commercial counterparts which has been offered to and used by Telecogresca. The Ticketing System presenter here is a production ready application which has proven its usefulness and robustness several times during stressful real life events.

Still, we are a bunch of perfectionists and there is always room for improvement. I have multiple ideas for the future development for this project, and a lot of them are already "Work in Progress".

In order to write this document, it was necessary for me to make what would be called a "feature freeze". This has allowed to present this document as a consistent view of the project. But the reality is that there is a lot of ongoing work, and different features (not explained previously) have been explored or at least considered. In this chapter, I will present both the ongoing work and the future work. I will present both with different level of abstraction, but I won't go into deep details on neither of them, because even being relevant for the global scenario, their inner workings are outside the scope of this document.

Ongoing work

New language, new framework

In this document I have presented the PHP-based web application –mainly during chapter 3. Truth is that, lately, I have been advocating for an unification of web applications and we have migrated onto a different language (Python) with a different framework (Django).

The internal structure remains the same, but doing the framework migration has allowed us to integrate more tightly the ticket management with other tools of Telecogresca Intranet, like I will explain in following points.

"Matxaca" management

Related to the prior feature, we have been working on a "Matxaca"[1] management Web Application.

Prior to the aforementioned management application and the integrated ticketing system, the credentials of those volunteers had to be checked by hand or through cumbersome spreadsheets. Now, the ticketing system allows us to use ticket-like validation, and the integration between the "Matxaca" management and the application system presents a unified interface.

This is becoming a key feature for the ongoing year's (2016) Telecogresca edition. I expect an almost full integration between the management application and the ticketing infrastructure. The (condensed) use case is the following:

1. A "*Reunió de Captació*" (recruitment meeting) is done, and all the personal data of the "Matxaques" is stored in the application.
2. The different tasks are assigned. Those tasks have been previously inserted into the application, and the application ensures that each task is satisfactorily assigned.
3. When the tasks are performed, they are marked as such into the application.

[1] This is the term that has been used, for historical reasons, to refer to the volunteers who participate with the Telecogresca in one-time tasks. Thanks to that manpower is possible to prepare the Telecogresca, but they do not participate continuously in the organization.

4. An internal hook ensures that a *Ticket* (referring to the ticketing application) is generated. The "Matxaca" receives their ticket for the festival. A certain relation is held between the "Matxaca" and the *Ticket*.

5. If there is any problem, the *Ticket* must be invalidated. This should be done through the "Matxaca" management application. Performing this "*Reward* invalidation" automatically triggers a hook which disables its associated *Ticket*.

Statistics

Having our own sale platform allows access to a lot information regarding the buyers. This information would be somewhat accessible even when using third party platforms. Also, by using external APIs like the Google Analytics [8], we would have access to those statistics –in fact, we already use them for related purposes. To take advantage of the available data, we prepared our own real-time statistics briefing, as an extra tool in our Intranet.

In addition to allow a real-time analysis of the sales –useful for evaluation of marketing campaigns, and to obtain briefings of the current state, like can be seen in figure 6.1– it serves as a historical record and comparison baseline, like shown in figure 6.2.

Future work

The Frontend Validation (see section 4.1) has a wide range of different options. With some supply of man-hours and/or resources, we could explore adapting some kind of turnstile validation. We have decided to not prioritize that because it would be a waste of resources, given that the infrastructure would be used in one single event per year.

Anecdotally, we have been unofficially "encouraged" by the Forum administrators to adapt our infrastructure to the turnstiles used by them. I did a preliminary exploration to see the viability of that endeavour. My conclusion was that it was well beyond our resources, and completely out of our personal interests[2]. I was (briefly) argued about my assessment, but they eventually let go the idea of turnstile validation.

Regarding the Frontend Validation, we also considered to use some kind of smart optical reader (like the one in figure 6.3). This would remove the need of laptops for the validation lanes. But requires a high commitment in both:

[2] I am a volunteer, I have done a lot of things to learn and because I wanted to. And because I can. Having a deadline for an uninteresting objective which would add no real value to the Ticketing Infrastructure does not sound like a good usage of my time.

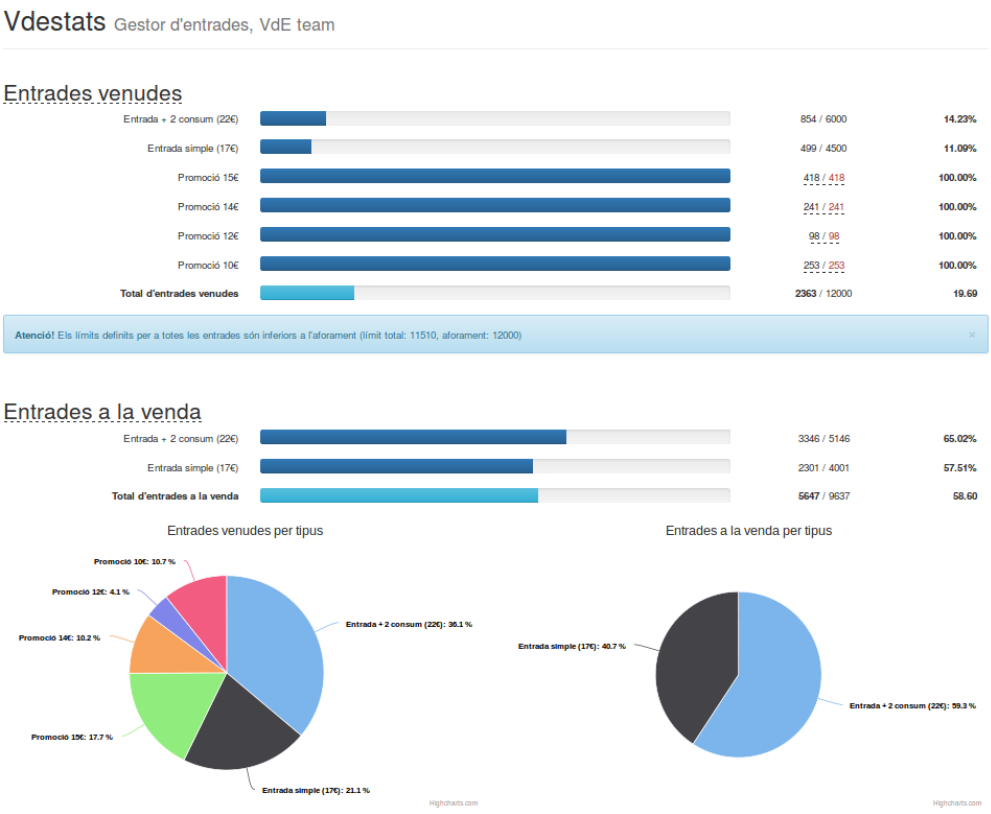


Figure 6.1: Briefing view

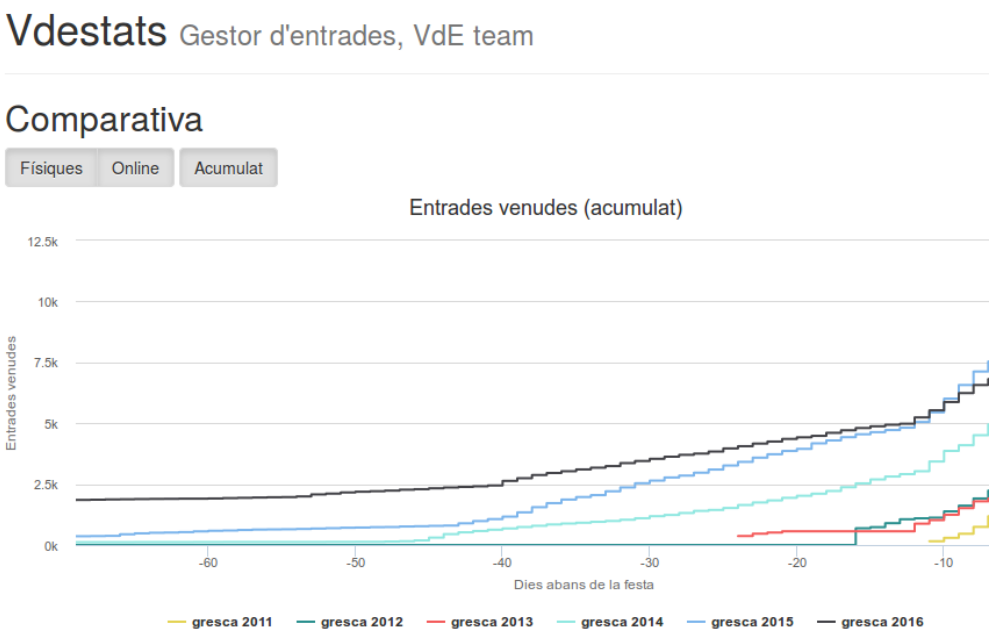


Figure 6.2: Comparison view

Application Development It requires to readapt the smartphone application (section 4.1.2) and deprecates the Qt application (section 4.1.3).

Monetary Resources Those devices are expensive. The price for a full-fledged scanner is similar to a consumer-grade laptop. Renting would avoid the need to buy and maintain the devices, but the price (somewhere near 100 € per unit per event) is still high for our budget, in addition to not being an investment.

I have already discussed in the "Ongoing Work" (previous section) the language migration to Python regarding the web framework. It may become advisable to also migrate the desktop application (section 4.1.3) to the Python language. This change would not be as bad as it sounds, in terms of development effort, because of PyQt [9]. This software package, not affiliated to Qt Company, provides language bindings to Qt for Python. Some code should be translated from C++ to Python, but all the structure and user interface layouts can be left unmodified. If this language is changed, all the software related to the ticketing infrastructure will be unified, lowering the learning curve for future developers.

Figure 6.3: Example of typical Android 2D Barcode Scanner

APPENDIX

A

DESKTOP APPLICATION DETAILS

In this chapter I present some further details and anecdotes regarding the desktop application first introduced in section 4.1.3. I will introduce relevant details on the design and development while trying to retain a high-level perspective.

I have been the lead developer[1] of this specific application and during this appendix I will give insight of major roadmap points that were either tricky, critical, complex or have some kind of overall interest worth mentioning.

Panic-mode scenarios

There are a handful of scenarios where we require a certain resilience from the application. We like to give a dramatic touch and call them "panic mode", and sometimes we refer their activation as "pressing the panic button". There are two scenarios that have been considered with great care:

Network or server failure From the point of view of the client, the behaviour is the same for both failures: the API does not give responses. This can be caused by a

[1] *Lead developer* is a fancy way of saying that I have done all the coding while my colleagues have helped me to bug-hunt, improve, field-test and brainstorm it.

power outage –although the deployment includes UPSs–, some cable breaking, a misconfiguration of the network... The solution implemented has been to include a hot backup in each PC, at an application level. This allows the system to run quite well in a degraded state. The drawback would be that double validations across different computers during the degraded state are not detected.

Unacceptable lag We envisioned that there might be situations in which the deployment performance was not satisfactory enough. Maybe the server would not handle too well a peak in the attendees incoming flow, and the responses would start to lag. The performance has generally been very good and the network latency has been pretty much constant, but the scenario was plausible. The "panic button" for this scenario changes the application behaviour and makes it trigger green flag *before* checking with the server –although after checking the local hot backup. The ticket is validated to the server, but the response is out of the critical path. There are certain race conditions that may arise, but the "damage control" is reasonable.

Multi-keyboard support

Typical generic barcode scanners behave as keyboards. See figure A.1 as a typical unbranded generic handheld barcode scanner. From the Operating System point of view, whenever a barcode is *read* by the handheld device, several keystrokes are triggered. This also means that reading some kinds of barcodes –like the Code 128 [4] ones– are keyboard-layout dependent.

Behaving as a keyboard is not a strong technological constraint of the "concept" of barcode scanners, but it is the most standard interface. On top of that, we experimented with several barcode scanner providers, and discovered that they tend to be the same generic board and some more or less fancy packaging and rebranding. For those reasons we decided that it would be an affordable idea to adapt to a standard notion of barcode scanner.

A simple solution would be to have one laptop per lane, and use one barcode handheld device per laptop. It is a trivial solution for organization that can afford that many laptops or that few lanes –for instance *Primavera Sound* has used this set up in previous editions and it has proved a good decision for them. However, Telecogresca has high flow peaks and not much hardware resources, so having so many laptops was not a good idea. This drove us to design the hardware prototype explained in 5.3 and do some field-tests.

Figure A.1: Generic barcode scanner

Our decision and current implementation at this moment[2] is centered on the `Xinput 2` [10] X11 extension.

Using this technology forces us to use GNU/Linux –which is what I was going to use nevertheless– and requires to use some X11 libraries without fancy encapsulation –the Qt framework offers encapsulation for most of the X11 quirks, but the `Xinput 2` extension is not amongst them.

The result of using this X11 extension is that the application is capable to recognize and discern different keyboards devices, which results on the ability to control from a single laptop several barcode scanners. We had considered this our goal, and I was able to implement the code to achieve that and has proven to work reliably under both the Raspberry Pi devices and regular consumer-grade laptops.

On the downside, the programmer must make some low-level management of events, like `KeyPress` and `KeyRelease` events. Things like alphanumeric case-sensitive barcodes (like the ones we are currently using) become a little bit tricky because of the case and the "Shift" key events, which should be tracked in order to correctly detect the case of the alphabetic characters.

[2] Last major event organized by Telecogresca, at the time of writing this document, which is Telecogresca 2016.

Network communication

The validation application can be considered a *client* to the validation backend, from the network standpoint. The design decision planning for this matter started before deciding Qt as the main programming framework for the project.

Once we chose the path of using C++ and the Qt framework, I considered the characteristics of the Qt network stack [11, 12] and decided that it was a good choice to explore. Given that Qt was already a dependency for the project, using the Qt network libraries is a sensible inclusion. On top of that, both the performance and the programmability soon proved to be adequate. I had to struggle with some details, but the overall experience was quite painless and the network communication worked reliably for all the tests that we performed.

Application Settings

Using some kind of settings container is a must-have for almost all applications. I am ashamed to say that the first versions of this application did *not* include said container –the settings were engraved in the application through a header file. This first *hardcoded settings* (sorry for the oxymoron) included, for instance, the backend URL endpoints for the ticket validation.

Once I gained some experience and insight for this matter, I migrated those options to a real settings file (using the Qt standard `QSettings` class). This settings file includes the previous URLs, but also contains more information like the *enabled* flags for each, or the local database path. Here you can see an example of a `SCA.conf` configuration file:

```
[global]
batch_url = http://172.19.0.2/entrades/api/
db_name = "database.sqlite"
race_mode = 0

[tickets]
T/url = http://172.19.0.2/entrades/api/
T/default_enabled = 1
T/description = "Entrades promocionals"
T/inclou = "Got"
G/url = http://172.19.0.2/entrades/api/
G/default_enabled = 1
G/description = "Entrades 21e"
```

```
G/inclou = "Got + Consumicions"
K/url = http://172.19.0.2/invites/api/
K/default_enabled = 0
K/description = "Invitacions"
K/inclou = "Res (invita)"
M/url = http://172.19.0.2/entrades/api/
M/default_enabled = 1
M/description = "Entrades 16e"
M/inclou = "Res (entrada simple)"
```

Once I finished implementing those settings mechanism, I realized that a hot refresh of those settings was a nice feature, and implemented it through the usage of the `SIGHUP` (a specific POSIX signal [13]). This is not an arbitrary choice: it is the preferred mechanism used by POSIX *daemons* to refresh (re-read) their configuration files, so it was a natural mechanism to add to our application.

Once that settings file and this "hot refresh" mechanism were included in the application, I was able to remotely update the settings and force an update by issuing a POSIX signal in each laptop to the running instance of the application. All this could be done manually through `ssh` or by using a more versatile tool like *Fabric* [14].

APPENDIX

B

YII PHP FRAMEWORK

Yii is a mature, free and open-source PHP framework with a rich set of features. According to its own documentation [15]:

> Yii is a high performance, component-based PHP framework for rapidly developing modern Web applications. (...) Yii is a generic Web programming framework, meaning that it can be used for developing all kinds of Web applications using PHP.

In the context of this project, the Yii framework has been used as the framework for the web application (Chapter 3).

The logo can be seen in figure B.1 and a short summary is available in table B.1.

Figure B.1: Yii logo

First alpha release	October 2006
First stable release	December 2008 (1.00)
Last stable release	February 2016 (2.0.7)
Written in	PHP
License	New BSD License
Website	www.yiiframework.com

Table B.1: Yii Software Summary

About Web Frameworks

A **Web Framework** is a piece of software that simplifies and enhances the task of developing web applications. A lay person may think that developing a web application consists on choosing a programming language and starting to develop the whole Web Application from scratch. This would inevitably result in reinventing the wheel many times. The correct course of action is to choose a Web Framework instead. By choosing a suitable Web Framework the development effort of the web application greatly decreases (from the imaginary situation of doing it from scratch), and we ensure a high final product quality.

The "logic" of the application is programmed in the chosen programming language (it often matches the framework language). The organization of code, the structure, and all the high level features vary from framework to framework. Each framework has an unique flavour and expects certain coding patterns and focus, in addition to encourages a way of doing things.

Yii key features

A huge list of Yii features can be found. I will highlight the most relevant features for this specific project and describe their strengths:

Model-View-Controller (MVC) design pattern Yii encourages the developer to use the Model-View-Controller, which has a lot of advantages over a non-structured programming style. This helps to have maintainable code and reduces the quantity of boiler-plate code for typical applications.

Database Access Objects (DAO) One error-prone and tedious task in web development is the management of database registries. The DAO presents a simple

abstraction to access the database without having to specialize all the queries and data access. This abstraction is key for obtaining a clean and valid data model, which can be efficiently used in the MVC pattern.

Form input and validation The typical web application will include forms to allow user-side data input. The data should be validated server-side and may be stored to the database. Yii includes patterns to cleanly implement the data validation and easy mechanisms to integrate the forms with the model. Having clean code for the forms reduces the risk of introducing validation errors and also reduces the coding effort for the developer.

Error handling and logging Applications are developed, debugged, broken and bugfixed in a cyclic fashion. The Yii framework includes extensive mechanisms for error handling, and those are available in both `DEBUG` and production environments. The logging mechanisms provide all the required information for typical scenarios.

Security The Yii framework has been extensively tested and is a secure framework. It has a long history of stable releases and a strong core developer team.

Extension library capabilities Sometimes the programmer needs some generic feature that feels very common but is not built-in inside Yii. For those situations, there are a lot of third-party libraries that can easily be added to a web application. Avoid reinventing the wheel is a good practice which yields better code and reduces the error sources, as the extension libraries are typically widely used and extensively tested.

APPENDIX

C

QT SOFTWARE

Qt is a programming framework for C++, programmed in C++. The first release is from 1995, initially by Trolltech company. It is currently owned by its own company, Qt Company. In the context of this project, the Qt software has been used for the development of the desktop application (section 4.1.3).

The logo can be seen in figure C.1 and a short summary is available in table C.1.

The main selling points of Qt, aside their extensive set of features, are the following:

Figure C.1: Qt logo

First release	May 1995
Last stable release	March 2016 (5.6)
Written in	C++
License	Multi-licensed: Qt Commercial License GPL-3.0 LGPL-3.0 LGPL-2.1
Website	qt.io

Table C.1: Qt Software Summary

Cross-platform The same code can be compiled and runs over Windows, Linux, and several embedded architectures.

Open Source It can be used under several licenses, and there is a lot of community support available. In our scenario, we can use it free of charge without any problem.

Developer tools There is an IDE available (see figures C.2 and C.3) which allows to easily design user interfaces and develop applications using the Qt framework.

Qt key features

I will not enumerate all the features and packages of the Qt framework as it would result in a huge list. Instead I list here some features used by my validation application. The module containing the feature can also be seen next to the feature itself.

- Threading [included in the main `QtCore`]
- Graphical User Interface [`QtGui`, `QtWidgets`]
- Network abstraction, supporting REST [`QtNetwork`]
- Local database (SQLite) [`QtSql`]
- JSON validation [`QJson*`]

Having all those features in a single framework reduces the learning curve –only one framework has to be learnt– and improves code consistency.

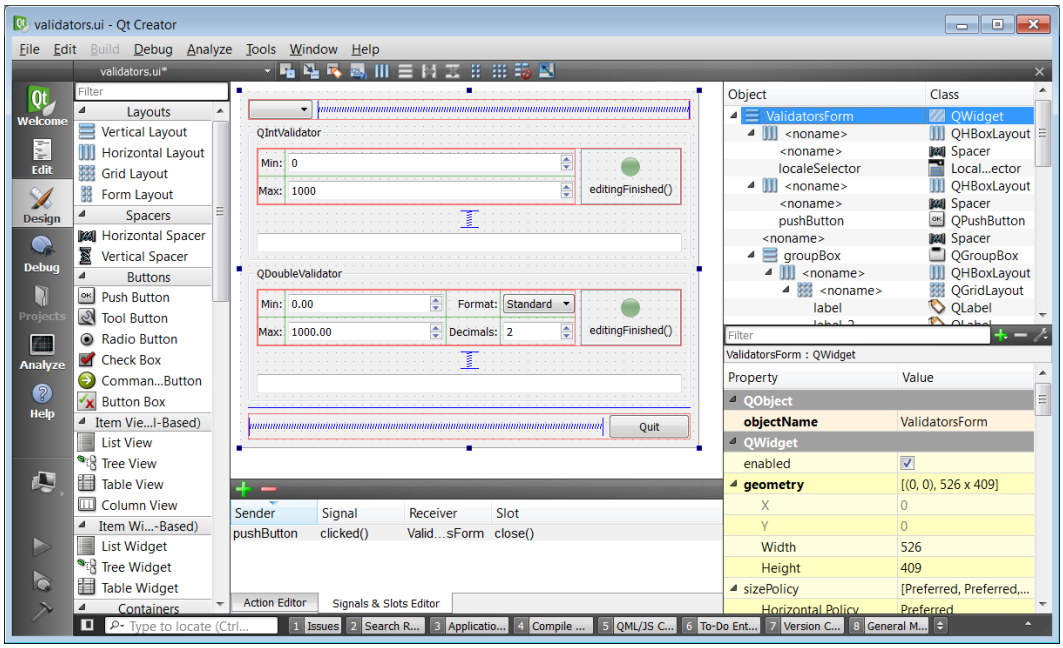


Figure C.2: Qt Creator (Integrated Development Editor)

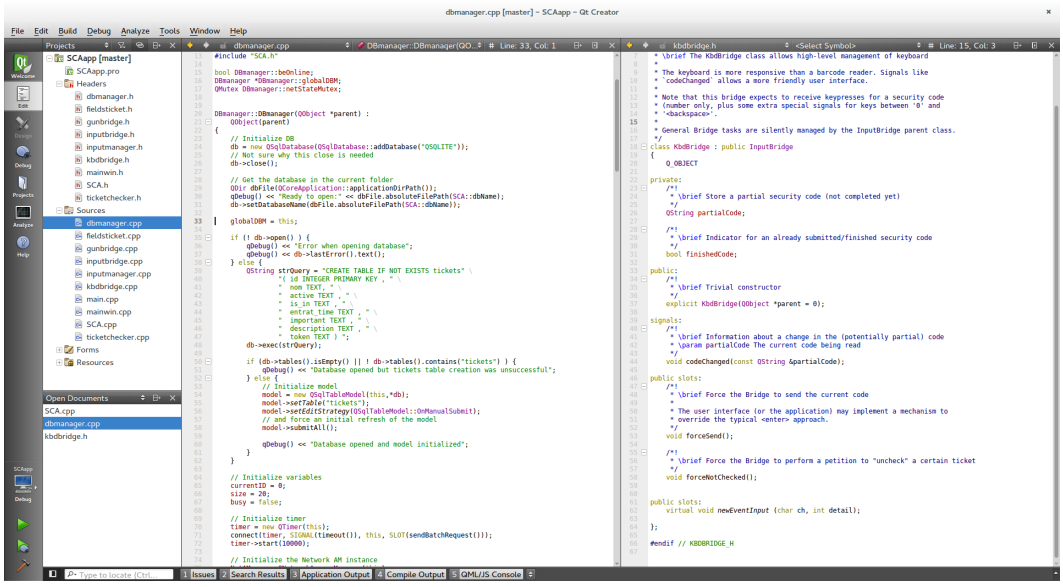


Figure C.3: C++ development with Qt Creator

BIBLIOGRAPHY

[1] Yii Framework. API Documentation. **December 2015**.
URL http://www.yiiframework.com/doc/api/

[2] Glenn E Krasner and Stephen T Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49. **1988**.

[3] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pp. 118–127. IEEE. **2001**.

[4] ISO/IEC 15417:2007. Code 128 bar code symbology specification. Technical report, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). **2012**.

[5] Digia Plc. Qt Documentation. **December 2015**.
URL http://doc.qt.io

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*, chapter 5 "Representational State Transfer (REST)". Ph.D. dissertation. **2000**.

[7] Raspberry Pi Foundation. Raspberry Pi. **December 2015**.
URL http://www.raspberrypi.org

[8] Google. Google Analytics. **January 2016**.
URL https://www.google.com/analytics/

[9] Riverbank Computing Limited. PyQt. **January 2015**.
URL https://riverbankcomputing.com/software/pyqt/intro

[10] X.Org Foundation. The X New Developer's Guide. **January 2015**.
URL http://www.x.org/wiki/guide/extensions/#index2h2

[11] Digia Plc. Qt Documentation – Network Programming (Qt 4.8). **December 2015**.
URL http://doc.qt.io/qt-4.8/network-programming.html

[12] Digia Plc. Qt Documentation – Network Programming with Qt (Qt 5). **January 2015**.
URL http://doc.qt.io/qt-5/qtnetwork-programming.html

[13] Linux man-pages project. SIGNAL(7): signal - overview of signals. Technical report, POSIX.1. **2014**.

[14] Jeff Forcier. Fabric documentation. **January 2016**.
URL http://www.fabfile.org/

[15] Yii Framework. The Definitive Guide to Yii 2.0. **February 2016**.
URL http://www.yiiframework.com/doc-2.0/guide-intro-yii.html